
MODULE *DistributedLock*

EXTENDS *Naturals, FiniteSets, Sequences, TLC*

The set of clients

CONSTANT *Clients*

Client states

CONSTANTS *Active, Inactive*

Message types

CONSTANT *LockRequest, LockResponse, TryLockRequest, TryLockResponse, UnlockRequest, UnlockResponse*

An empty constant

CONSTANT *Nil*

The current lock holder

VARIABLE *lock*

The lock queue

VARIABLE *queue*

The current lock *ID*

VARIABLE *id*

Session states

VARIABLE *sessions*

$serverVars \triangleq \langle lock, id, queue, sessions \rangle$

Client states

VARIABLE *clients*

$clientVars \triangleq \langle clients \rangle$

Client requests

VARIABLE *requests*

Server responses

VARIABLE *responses*

Variable to track the total number of messages sent for use in state constraints when model checking

VARIABLE *messageCount*

$messageVars \triangleq \langle requests, responses, messageCount \rangle$

The invariant checks that:

- * No client can hold more than one lock at a time
- * No two clients hold a lock with the same *ID*

* The lock is held by an active session

Note that more than one client may believe itself to hold the lock at the same time, *e.g.* if a client's session has expired but the client hasn't been notified, but lock *IDs* must be unique and monotonically increasing.

$TypeInvariant \triangleq$

$\wedge \forall c \in \text{DOMAIN } clients : Cardinality(clients[c].locks) \in 0 .. 1$

$\wedge \forall c1, c2 \in \text{DOMAIN } clients : c1 \neq c2 \Rightarrow Cardinality(clients[c1].locks \cap clients[c2].locks) = 0$

$\wedge lock \neq Nil \Rightarrow sessions[lock.client].state = Active$

Returns a sequence with the head removed

$Pop(q) \triangleq SubSeq(q, 2, Len(q))$

Sends a request on the given client's channel

$SendRequest(m, c) \triangleq$

$\wedge requests' = [requests \text{ EXCEPT } ![c] = Append(requests[c], m)]$

$\wedge messageCount' = messageCount + 1$

Sends a response on the given client's channel

$SendResponse(m, c) \triangleq$

$\wedge responses' = [responses \text{ EXCEPT } ![c] = Append(responses[c], m)]$

$\wedge messageCount' = messageCount + 1$

Removes a request from the given client's channel

$AcceptRequest(m, c) \triangleq$

$\wedge requests' = [requests \text{ EXCEPT } ![c] = Pop(requests[c])]$

$\wedge messageCount' = messageCount + 1$

Removes a response from the given client's channel

$AcceptResponse(m, c) \triangleq$

$\wedge responses' = [responses \text{ EXCEPT } ![c] = Pop(responses[c])]$

$\wedge messageCount' = messageCount + 1$

This section models a lock state machine. The state machine supports three types of request:

* *LockRequest* attempts to acquire the lock. If the lock is owned by another process, the request is enqueued until the lock is released.

* *TryLockRequest* attempts to acquire the lock and fails if the lock is already owned by another process.

* *UnlockRequest* attempts to release a lock that's owned by a process.

Additionally, any process's session can be expired, causing any locks held by the session to be released and lock requests enqueued for the session to be removed.

Handles a lock request. If the lock is not currently held by another process, the lock is granted to the client. If the lock is held by a process, the request is added to a queue.

$HandleLockRequest(m, c) \triangleq$

$$\begin{aligned}
& \vee \wedge \text{sessions}[c].\text{state} \neq \text{Active} \\
& \quad \wedge \text{UNCHANGED } \langle \text{clientVars}, \text{serverVars}, \text{responses} \rangle \\
& \vee \wedge \text{sessions}[c].\text{state} = \text{Active} \\
& \quad \wedge \text{lock} = \text{Nil} \\
& \quad \wedge \text{lock}' = m @ @ (\text{"client"} :> c) \\
& \quad \wedge \text{id}' = \text{id} + 1 \\
& \quad \wedge \text{SendResponse}([\text{type} \mapsto \text{LockResponse}, \text{acquired} \mapsto \text{TRUE}, \text{id} \mapsto \text{id}'], c) \\
& \quad \wedge \text{UNCHANGED } \langle \text{queue}, \text{sessions}, \text{clientVars} \rangle \\
& \vee \wedge \text{sessions}[c].\text{state} = \text{Active} \\
& \quad \wedge \text{lock} \neq \text{Nil} \\
& \quad \wedge \text{queue}' = \text{Append}(\text{queue}, m @ @ (\text{"client"} :> c)) \\
& \quad \wedge \text{UNCHANGED } \langle \text{lock}, \text{id}, \text{sessions}, \text{clientVars}, \text{responses} \rangle
\end{aligned}$$

Handles a *tryLock* request. If the lock is not currently held by another process, the lock is granted to the client. Otherwise, the request is rejected.

$$\begin{aligned}
& \text{HandleTryLockRequest}(m, c) \triangleq \\
& \quad \vee \wedge \text{sessions}[c].\text{state} \neq \text{Active} \\
& \quad \quad \wedge \text{UNCHANGED } \langle \text{clientVars}, \text{serverVars}, \text{responses} \rangle \\
& \quad \vee \wedge \text{sessions}[c].\text{state} = \text{Active} \\
& \quad \quad \wedge \text{lock} = \text{Nil} \\
& \quad \quad \wedge \text{lock}' = m @ @ (\text{"client"} :> c) \\
& \quad \quad \wedge \text{id}' = \text{id} + 1 \\
& \quad \quad \wedge \text{SendResponse}([\text{type} \mapsto \text{LockResponse}, \text{acquired} \mapsto \text{TRUE}, \text{id} \mapsto \text{id}'], c) \\
& \quad \quad \wedge \text{UNCHANGED } \langle \text{queue}, \text{sessions}, \text{clientVars} \rangle \\
& \quad \vee \wedge \text{sessions}[c].\text{state} = \text{Active} \\
& \quad \quad \wedge \text{lock} \neq \text{Nil} \\
& \quad \quad \wedge \text{SendResponse}([\text{type} \mapsto \text{LockResponse}, \text{acquired} \mapsto \text{FALSE}], c) \\
& \quad \quad \wedge \text{UNCHANGED } \langle \text{clientVars}, \text{serverVars} \rangle
\end{aligned}$$

Handles an unlock request. If the lock is currently held by the given client, it will be unlocked. If any client's requests are pending in the queue, the next lock request will be removed from the queue and the lock will be granted to the requesting client.

$$\begin{aligned}
& \text{HandleUnlockRequest}(m, c) \triangleq \\
& \quad \vee \wedge \text{sessions}[c].\text{state} \neq \text{Active} \\
& \quad \quad \wedge \text{UNCHANGED } \langle \text{clientVars}, \text{serverVars}, \text{responses} \rangle \\
& \quad \vee \wedge \text{sessions}[c].\text{state} = \text{Active} \\
& \quad \quad \wedge \text{lock} = \text{Nil} \\
& \quad \quad \wedge \text{UNCHANGED } \langle \text{clientVars}, \text{serverVars}, \text{responses} \rangle \\
& \quad \vee \wedge \text{sessions}[c].\text{state} = \text{Active} \\
& \quad \quad \wedge \text{lock} \neq \text{Nil} \\
& \quad \quad \wedge \text{lock.client} = c \\
& \quad \quad \wedge \text{lock.id} = m.\text{id} \\
& \quad \quad \wedge \vee \wedge \text{Len}(\text{queue}) > 0 \\
& \quad \quad \quad \wedge \text{LET } \text{next} \triangleq \text{Head}(\text{queue}) \\
& \quad \quad \quad \text{IN} \\
& \quad \quad \quad \wedge \text{lock}' = \text{next}
\end{aligned}$$

$$\begin{aligned}
& \wedge id' = id + 1 \\
& \wedge queue' = Pop(queue) \\
& \wedge SendResponse([type \mapsto LockResponse, acquired \mapsto TRUE, id \mapsto id'], next.client) \\
& \wedge UNCHANGED \langle sessions \rangle \\
\vee & \wedge Len(queue) = 0 \\
& \wedge lock' = Nil \\
& \wedge UNCHANGED \langle queue, id, sessions, responses \rangle \\
& \wedge UNCHANGED \langle clientVars \rangle
\end{aligned}$$

Expires a client's session. If the client currently holds the lock, the lock will be released and the lock will be granted to another client if possible. Additionally, pending lock requests from the client will be removed from the queue.

$$\begin{aligned}
& ExpireSession(c) \triangleq \\
& \wedge sessions[c].state = Active \\
& \wedge sessions' = [sessions \text{ EXCEPT } ![c].state = Inactive] \\
& \wedge LET isActive(m) \triangleq sessions'[m.client].state = Active \\
& IN \\
& \quad IF lock \neq Nil \wedge lock.client = c THEN \\
& \quad \quad LET q \triangleq SelectSeq(queue, isActive) \\
& \quad \quad IN \\
& \quad \quad \vee \wedge Len(q) > 0 \\
& \quad \quad \quad \wedge lock' = Head(q) \\
& \quad \quad \quad \wedge id' = id + 1 \\
& \quad \quad \quad \wedge queue' = Pop(q) \\
& \quad \quad \quad \wedge SendResponse([type \mapsto LockResponse, acquired \mapsto TRUE, id \mapsto id'], lock'.client) \\
& \quad \quad \quad \wedge UNCHANGED \langle requests \rangle \\
& \quad \vee \wedge Len(queue) = 0 \\
& \quad \quad \wedge lock' = Nil \\
& \quad \quad \wedge queue' = \langle \rangle \\
& \quad \quad \wedge UNCHANGED \langle id, messageVars \rangle \\
& \quad ELSE \\
& \quad \quad \wedge queue' = SelectSeq(queue, isActive) \\
& \quad \quad \wedge UNCHANGED \langle lock, id, messageVars \rangle \\
& \wedge UNCHANGED \langle clientVars \rangle
\end{aligned}$$

This section models a lock client. A client can interact with a lock state machine using three types of requests:

- * *Lock* attempts to acquire a lock and blocks until successful or the session expires
- * *TryLock* attempts to acquire a lock, failing if the lock is owned by another process
- * *Unlock* attempts to release a lock owned by the process

Additionally, a client can assume its session has expired either before or after it actually has. This models the possibility that a client believes its session has expired when it hasn't or that the state machine can expire a client's session without the client knowing.

Sends a lock request to the cluster with a unique *ID* for the client.

$$\begin{aligned}
\text{Lock}(c) &\triangleq \\
&\wedge \text{clients}[c].\text{state} = \text{Active} \\
&\wedge \text{SendRequest}([type \mapsto \text{LockRequest}, id \mapsto \text{clients}[c].\text{next}], c) \\
&\wedge \text{clients}' = [\text{clients} \text{ EXCEPT } ![c].\text{next} = \text{clients}[c].\text{next} + 1] \\
&\wedge \text{UNCHANGED } \langle \text{serverVars}, \text{responses} \rangle
\end{aligned}$$

Sends a try lock request to the cluster with a unique *ID* for the client.

$$\begin{aligned}
\text{TryLock}(c) &\triangleq \\
&\wedge \text{clients}[c].\text{state} = \text{Active} \\
&\wedge \text{SendRequest}([type \mapsto \text{TryLockRequest}, id \mapsto \text{clients}[c].\text{next}], c) \\
&\wedge \text{clients}' = [\text{clients} \text{ EXCEPT } ![c].\text{next} = \text{clients}[c].\text{next} + 1] \\
&\wedge \text{UNCHANGED } \langle \text{serverVars}, \text{responses} \rangle
\end{aligned}$$

Sends an unlock request to the cluster if the client is active and current holds a lock.

$$\begin{aligned}
\text{Unlock}(c) &\triangleq \\
&\wedge \text{clients}[c].\text{state} = \text{Active} \\
&\wedge \text{Cardinality}(\text{clients}[c].\text{locks}) > 0 \\
&\wedge \text{SendRequest}([type \mapsto \text{UnlockRequest}, id \mapsto \text{CHOOSE } l \in \text{clients}[c].\text{locks} : \text{TRUE}], c) \\
&\wedge \text{clients}' = [\text{clients} \text{ EXCEPT } ![c].\text{locks} = \text{clients}[c].\text{locks} \setminus \{\text{CHOOSE } l \in \text{clients}[c].\text{locks} : \text{TRUE}\}] \\
&\wedge \text{UNCHANGED } \langle \text{serverVars}, \text{responses} \rangle
\end{aligned}$$

Handles a lock response from the cluster. If the client's session is expired, the response is ignored. If the lock was acquired successfully, it's added to the client's lock set.

$$\begin{aligned}
\text{HandleLockResponse}(m, c) &\triangleq \\
&\wedge \vee \wedge \text{clients}[c].\text{state} = \text{Inactive} \\
&\quad \wedge \text{UNCHANGED } \langle \text{clientVars}, \text{serverVars}, \text{requests} \rangle \\
&\vee \wedge \text{clients}[c].\text{state} = \text{Active} \\
&\quad \wedge m.\text{acquired} \\
&\quad \wedge \text{clients}' = [\text{clients} \text{ EXCEPT } ![c].\text{locks} = \text{clients}[c].\text{locks} \cup \{m.\text{id}\}] \\
&\quad \wedge \text{UNCHANGED } \langle \text{serverVars}, \text{requests} \rangle \\
&\vee \wedge \text{clients}[c].\text{state} = \text{Active} \\
&\quad \wedge \neg m.\text{acquired} \\
&\quad \wedge \text{UNCHANGED } \langle \text{clientVars}, \text{serverVars}, \text{requests} \rangle
\end{aligned}$$

Closes a client's expired session. This is performed in a separate step to model the time between the cluster expiring a session and the client being notified. A client can close its session either before or after it's expired by the cluster. Once the client believes its session has expired, its locks are removed, meaning a client can also believe itself to hold a lock after its session has expired in the cluster.

$$\begin{aligned}
\text{CloseSession}(c) &\triangleq \\
&\wedge \text{clients}[c].\text{state} = \text{Active} \\
&\wedge \text{clients}' = [\text{clients} \text{ EXCEPT } ![c].\text{state} = \text{Inactive}, \\
&\quad \quad \quad ![c].\text{locks} = \{\}] \\
&\wedge \text{UNCHANGED } \langle \text{serverVars}, \text{messageVars} \rangle
\end{aligned}$$

Receives a request 'm' from the client 'c' to the cluster.

$$\begin{aligned}
\text{ReceiveRequest}(m, c) &\triangleq \\
&\wedge \vee \wedge m.type = \text{LockRequest} \\
&\quad \wedge \text{HandleLockRequest}(m, c) \\
&\vee \wedge m.type = \text{TryLockRequest} \\
&\quad \wedge \text{HandleTryLockRequest}(m, c) \\
&\vee \wedge m.type = \text{UnlockRequest} \\
&\quad \wedge \text{HandleUnlockRequest}(m, c) \\
&\wedge \text{AcceptRequest}(m, c)
\end{aligned}$$

Receives a response 'm' from the cluster to the client 'c'.

$$\begin{aligned}
\text{ReceiveResponse}(m, c) &\triangleq \\
&\wedge \vee \wedge m.type = \text{LockResponse} \\
&\quad \wedge \text{HandleLockResponse}(m, c) \\
&\wedge \text{AcceptResponse}(m, c)
\end{aligned}$$

Initial state predicate

$$\begin{aligned}
\text{Init} &\triangleq \\
&\wedge \text{requests} = [c \in \text{Clients} \mapsto \langle \rangle] \\
&\wedge \text{responses} = [c \in \text{Clients} \mapsto \langle \rangle] \\
&\wedge \text{messageCount} = 0 \\
&\wedge \text{lock} = \text{Nil} \\
&\wedge \text{queue} = \langle \rangle \\
&\wedge \text{id} = 0 \\
&\wedge \text{clients} = [c \in \text{Clients} \mapsto [\text{state} \mapsto \text{Active}, \text{locks} \mapsto \{\}, \text{next} \mapsto 1]] \\
&\wedge \text{sessions} = [c \in \text{Clients} \mapsto [\text{state} \mapsto \text{Active}]]
\end{aligned}$$

Next state predicate

$$\begin{aligned}
\text{Next} &\triangleq \\
&\vee \exists c \in \text{DOMAIN } \text{clients} : \exists i \in \text{DOMAIN } \text{requests}[c] : \text{ReceiveRequest}(\text{requests}[c][i], c) \\
&\vee \exists c \in \text{DOMAIN } \text{clients} : \exists i \in \text{DOMAIN } \text{responses}[c] : \text{ReceiveResponse}(\text{responses}[c][i], c) \\
&\vee \exists c \in \text{DOMAIN } \text{clients} : \text{Lock}(c) \\
&\vee \exists c \in \text{DOMAIN } \text{clients} : \text{TryLock}(c) \\
&\vee \exists c \in \text{DOMAIN } \text{clients} : \text{Unlock}(c) \\
&\vee \exists c \in \text{DOMAIN } \text{clients} : \text{ExpireSession}(c) \\
&\vee \exists c \in \text{DOMAIN } \text{clients} : \text{CloseSession}(c)
\end{aligned}$$

The specification includes the initial state predicate and the next state

$$\text{Spec} \triangleq \text{Init} \wedge \Box[\text{Next}]_{\langle \text{serverVars}, \text{clientVars}, \text{messageVars} \rangle}$$

\ * Modification History
\ * Last modified Sun Jan 28 12:14:41 PST 2018 by jordanhalterman
\ * Created Fri Jan 26 13:12:01 PST 2018 by jordanhalterman